1 Problem 2(a)

// Shared variables
int n;
int _iplist[n], _pprefixsum[n], _pairwisesum[n];

// Function to create the threads
void parallel_prefix()
{
    int i;
    for (i=0; i < (n/2); i++) {
        // create n/2 threads
        thread_create(parallel_prefix_thread, i);
    }
}

// Function executed by each thread
void parallel_prefix_thread(int tid)
{
    // local variables
    int i, _index, _lchild, _rchild;

    // marks where val needs to be read or written
    _index = 0;

    // points to read array for pairwise sum
    int *ptr;
    ptr = _iplist;
    for (i = n; i >=1; ) {
        // control the thread execution based on id
        if (tid < i) {
            if (_index != 0) {
                // index calculations to read from intermediate results
                _lchild = (_index - i) + (2 * tid) + 0;
                _rchild = _lchild + 1;
            }
        } else {
            // index calculations to read from iplist
            _lchild = 2 * tid;
            _rchild = 2 * tid + 1;
        }
        // pairwise summation adding left and right

}
pairwisesum[\_index+tid] = ptr[\_lchild] + ptr[\_rchild];

\_index += \_i;
ptr = \_pairwisesum;
i = i/2;

// initialize to 0
memset(\_pprefixsum, 0, sizeof(int) * \_n); // ignore the root value

for (i = 1; i < = \_n/2; ) {
    if (tid < i) {
        \_lchild = \_pprefixsum[tid];
        if(index != 0) {
            \_index = \_index - 2 * \_i;
            \_rchild = \_pairwisesum[\_index + (2 * tid)];
        } else {
            \_lchild += \_iplist[2*tid]
            \_rchild = \_iplist[2*tid]+1;
        }

        // barrier to ensure no race between _pprefixsum read & write
        barrier;

    } else {
        \_lchild *= 2;
    }
}

// prefix sum calculation
\_pprefixsum[2*tid] = \_lchild;
\_pprefixsum[2*tid+1] = \_lchild + \_rchild;
2 Problem 2(b)

For sorting, similar to parallel pair-wise sum we can do a upward sweep. Instead of adding the elements, we can merge them to have sorted sublists and proceed up to the root to finally merge two n/2 sorted sublists.

3 Problem 3

Original Loop :

```c
for (j = 0; j < N; j++)
    for (i = 0; i < N; i++)
        a[i+1][j+1] = a[i][j] + c;
```

There is a true dependency between $a[i][j]$ and $a[i+1][j+1]$ across different iterations in $i, j$.

Transformed Loop :

```c
for (j=0; j<N; j++)
    for (i=N-1; i>=0; i--)
        a[i+1][j+1] = a[i][j] + c;
```

Loop reversal transformation for the given code does not reverse the dependency from true to anti. This is because the reversed $i$ loop only reorders the execution of statements within $j$ iteration or in other words the outer loop carries the dependency across the iterations and hence it preserves the dependency. This result can also be shown by unrolling the loop for few iterations of $i,j$.